COMP26120
Academic Session: 2018-19

Lab Exercise 9: Graph Representation and Traversal

Duration: 1 lab session

For this lab exercise you should do all your work in your COMP26120/ex9 directory. Copy the starting files from */opt/info/courses/COMP26120/problems/ex9*.

For this assignment, you can only get full credit if you do not use code from outside sources.

1 Learning Outcomes

By the end of this lab you should be able to:

- Develop a data structure to represent directed graphs using adjacency lists.
- Implement depth-first, breadth-first traversal algorithms.
- Explain the advantages of adjacency lists over adjacency matrix representation.

2 Background

Directed graph

A directed graph is a set of vertices connected by edges, where the edges have a direction that points from a source node to a target node.

Adjacency list representation

An adjacency list representation of a graph consists of a list of all nodes, and with each node n a list of all adjacent nodes (for directed graphs, these are the nodes that are the target of edges with source n). An example is given below in figure 1.

out-degree and in-degree

The out-degree of a vertex is the number of edges directed out of that vertex in a directed graph. The in-degree of a vertex is the number of edges directed into that vertex in a directed graph

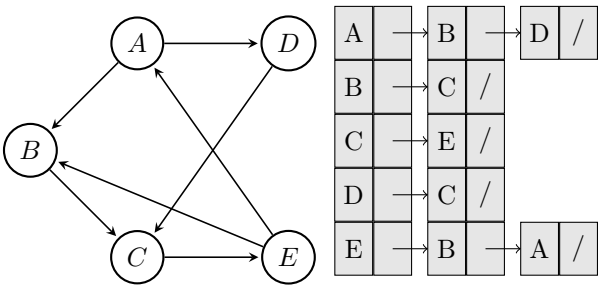

Figure 1: A directed graph and its representation in adjacency list.

breadth-first search implementation

For breadth-first search, you need to maintain a search queue and a list of explored nodes. To carry out a step of search, you

1. get and remove the first node from the search queue and process it;
2. get its adjacency list, and check each node to see if it is in the explored list;
3. If it is not, add it to the back of the queue, and to the explored list.

When the search list is empty, the algorithm will have explored all the nodes reachable from the starting node.

depth-first search implementation

In the above description of breadth-first search, the search list is a **queue**; you remove from the front and add to the back, i.e., it is first-in first-out (FIFO). If you replace the search list by **stack** in which you remove from the front and add to the front (FILO), the traversal becomes depth-first search. This is a non-recursive implementation.

You can also use the recursive algorithm that

1. starts from the given node;
2. get its adjacency list, for each node that is not explored before, recursively call dfs function;

3 Description

The data structure and the algorithm you develop in this lab will be used and extended in the next lab, so it is important that you do this lab on time in order for you to be able to carry on with the later labs.

To get you started, we will give you some data files containing descriptions of directed graphs, and some code to read this into your data structure.

3.1 graph.h

Modify *graph.h* so it contains the definition of data type GRAPH.

A graph is a collection of nodes. A node is a structure which contains fields for the following:

- its name (a string);
- an adjacency list of indices of the nodes it has links pointing to (a linked list of int);
- its out-degree, which is the length of the adjacency list (an int) and a field for its pagerank score.

Not all of these fields are needed for this lab; some will be used in the next lab.

3.2 graph_functions.c

Modify *graph_functions.c* so the functions *initialize_graph*, *insert_graph_node* (add nodes) and *insert_graph_link* (add edges) are implemented. If you change the prototypes, be sure to change the calls in *read_graph*.

For this lab, the size of graph (the number of nodes it contains) will be given as the first piece of information about the graph. Thus, you know when the graph is initialized how much memory will be required.

3.3 part1.c

Produce *part1.c* to run through the graph looking for the following three nodes. Run *make part1* to make this.

- the node with the largest out-degree
- the node with the largest in-degree
- the node with the smallest (non-zero) in-degree

This part should be very easy, as you just need loop through the graph an appropriate number of times. **Think about, and be prepared to answer how these tasks would be different if we had used adjacency matrix representation of the graph.** Which would be easy or run faster. Which would be harder or run slower.

The output from running `./part1 small.gx` should be *exactly* as follows:

```
Reading graph from small.gx
Max out-degree is 5 at node 3
Max in-degree is 4 at node 6
Min in-degree is 2 at node 3
```

e.g. the above text should be used to report the value of the out/in-degree and the name of the relevant node. If more than one node has the same value then the node with the lowest index should be reported.

Clarification: You should print a nodes name, not its index.

3.4 part2.c

Produce a file *part2.c* that implements a depth-first search and a breadth-first search traversal. Each function should take as input a node index, and search all the nodes reachable from that node. The *main* function should then call the search code to

- find the node with the largest out-degree which can be reached from the node with the smallest (non-zero) out-degree;
- count the number of nodes reachable from the node with the smallest (non-zero) out-degree.

The first point is written as if it has a unique answer but this is not necessarily the case. If a graph contains more than one such node, then you can arbitrarily pick any one of them - e.g. the first such node that you find (which may be different whether you use breadth-first or depth-first search).

Clarifications: In the above it is necessary to iterate over all nodes to identify the node with the smallest (non-zero) out-degree. It is then necessary to use a search algorithm to identify nodes that are *reachable* from this node. Not all nodes are reachable from all other nodes (this is an important point with graphs). A node is reachable from itself.

Use the preprocessor along with the flags `BFS` and `DFS` to select which search algorithm to use e.g.

```
#ifdef BFS
.... breadth_first_search(....)
#endif
#ifdef DFS
.... depth_first_search(....)
#endif
```

Then the commands *make part2depth* and *make part2breadth* should compile the two alternative executables.

The output from running `./part2_depth small.gx` or `./part2_breadth small.gx` should be *exactly* as follows:

```
Reading graph from small.gx
The node with the smallest (non-zero) out-degree is
1 with out-degree 1
There are 5 nodes reachable from the node with the
smallest (non-zero) out-degree and the node with the
largest out-degree is 3 with out-degree 5
```

We use `small.gx` as it has a unique node with smallest (non-zero) out-degree and a unique node with largest out-degree reachable from this node. You should test your code on other graphs, and graphs you create yourself.

Consider how you would extend your code to answer the following questions:

1. Is the node with the largest out-degree reachable from the node with the smallest (non-zero) out-degree?
2. If not, which is the node with the smallest (non-zero) out-degree from which the node with the largest out-degree is reachable from?
3. Can the entire graph be reached from the node with the smallest (non-zero) out-degree?
4. Is it possible to say which is better for each of these tasks (including that of part 2 above), breadth-first or depth first search?

Note: the above questions have been slightly reworded.

Input

In the directory */opt/info/courses/COMP26120/problems/ex9/data* you will find files with the suffix .gx. These contain descriptions of graphs, using a simple format. Here is an example:

```
MAX 10
NODE 1 one
EDGE 1 2
EDGE 1 3
EDGE 1 1
NODE 3 two
EDGE 3 5
EDGE 3 1
NODE 5 three
EDGE 5 7
EDGE 5 1
NODE 7 four
EDGE 7 7
NODE 9 five
NODE 2 six
```

The first line "MAX 10" tells that the node will have indices 1-10. The first line of these graphical files will always have such a line. Lines of the form "NODE 1 name" defines node number 1 with name "name". Lines of the form "EDGE 3 5" define an edge from node 3 to node 5. So, in the above example, Node 1 has as its adjacency list (1,2,3) since there are edges from 1 to 1, 1 to 2 and 1 to 3. The order of these lines is unimportant except the MAX line must come first. We have given you code which reads in such files, which can be found in *graph_functions.c*.

It is a good idea to produce your own small graph files to test your code.

4 Marking Scheme

Have graph.h and graph_functions.c been completed correctly and sensibly? Can the student explain their code?	
The graph structure is completed along with the initialize_graph, insert_graph_node and insert_graph_link functions. The graph structure may assume a fixed number of nodes (but more sophisticated solutions are okay). The structure should not be redundant (edges should be recorded in one place only). The graph and node structures may be extended with additional information. The functions should not leak memory. The student can briefly explain their code; importantly, any changes to the given code can be justified.	(2)
As above but a small mistake not affecting the correctness of the code has been made e.g. a single memory leak or poorly designed structures.	(1.8)
As above but there is an edge case missing leading to incorrect behaviour in some cases.	(1.5)
The code is good but it cannot be explained.	(1)
A significant part of the code is incomplete or incorrect e.g. the adding of edges always fails.	(1)
An attempt has been made.	(0.5)
No attempt has been made	(0)

Has part1 been completed correctly and can the solution be explained?	
The correct answer is found. This does not need to be the most efficient solution but the student must be able to explain the complexity of the solution. The student can explain the meaning of in-degree and out-degree and what it means for this to be 0 in each case.	(2)
The correct answer is found and the understanding is good but the test fails due to incorrect formatting.	(1.8)
The correct answer is found but there is a small gap in understanding.	(1.8)
The correct answer is found but the explanation is insufficient.	(1.5)
The wrong answer may be found due to a small mistake e.g. forgetting to include the first or last node.	(1)
An attempt has been made.	(0.5)
No attempt has been made	(0)

Can the student explain the difference between the adjacency list and adjacency matrix representation and how this might impact the computation in part 1?	
A good answer is given to both parts e.g. the difference between the representations is clearly explained and the student correctly identifies how a different representation might affect the computation (it is not sufficient to just say it would be worse/better).	(1)
As above but there is a small gap in understanding.	(0.8)
The student can explain the difference between representations but not what impact this might have.	(0.5)
No attempt has been made	(0)

Has part2 been completed correctly and can the solution be explained?	
Both depth-first and breadth-first search are implemented correctly. The correct answer is found. The student can explain the difference between the two search approaches. There are no memory leaks	(3)
As above but the test fails due to incorrect formatting.	(2.8)
As above but there is a minor flaw in one algorithm e.g. the original node is not counted as reachable from itself, some nodes are visited multiple times, or there is a minor memory leak. A flaw is minor if the correct answer is still found.	(2.5)
As above but there is a more significant flaw in one algorithm e.g. some reachable nodes are not found.	(2)
As above but the explanation is insufficient.	(2)
Only one of depth-first or breadth-first search has been implemented but this is correct and explained well.	(2)
Only one algorithm has been implemented and it has a minor flaw.	(1.8)
There are major flaws in the implementation of both algorithms.	(1)
An attempt has been made but the solution is far from complete.	(0.5)
No attempt has been made	(0)

Can the student answer the further questions?	
The student engages with the questions, can identify the differences in the scenarios, and can reflect suitably on when the choice between depth-first and breadth-first matters.	(1)
As above with a reasonable amount of prompting.	(0.8)
The student struggles to answer the questions but makes an attempt.	(0.5)
No attempt has been made	(0)

Is the code of good quality? Note that the students are not expected to handle unexpected situations such as badly formatted graphs in this lab.	
The quality of the code is good throughout (reasonable names, sufficient comments, sensible layout).	(1)
As above but there are a few minor issues	(0.8)
As above but there are some significant issues e.g. zero comments, completely illogical names	(0.5)
The quality of the code is poor throughout	(0)